IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2023, No. 2, pp. 24–53. DOI:10.46586/tches.v2023.i2.24-53

# Peek into the Black-Box: Interpretable Neural Network using SAT Equations in Side-Channel Analysis

Trevor Yap[1], Adrien Benamira[2], Shivam Bhasin[3] and Thomas Peyrin[4]

School of Physical and Mathematical Sciences,
Nanyang Technological University, Singapore
{trevor.yap,sbhasin,thomas.peyrin}@ntu.edu.sg[1,3,4] adrien002@e.ntu.edu.sg[2]

**Abstract.** Deep neural networks (DNN) have become a significant threat to the security of cryptographic implementations with regards to side-channel analysis (SCA), as they automatically combine the leakages without any preprocessing needed, leading to a more efficient attack. However, these DNNs for SCA remain mostly black-box algorithms that are very difficult to interpret. Benamira *et al.* recently proposed an interpretable neural network called Truth Table Deep Convolutional Neural Network (TT-DCNN), which is both expressive and easier to interpret. In particular, a TT-DCNN has a transparent inner structure that can entirely be transformed into SAT equations after training. In this work, we analyze the SAT equations extracted from a TT-DCNN when applied in SCA context, eventually obtaining the rules and decisions that the neural networks learned when retrieving the secret key from the cryptographic primitive (i.e., exact formula). As a result, we can pinpoint the critical rules that the neural network uses to locate the exact Points of Interest (PoIs). We validate our approach first on simulated traces for higher-order masking. However, applying TT-DCNN on real traces is not straightforward. We propose a method to adapt TT-DCNN for application on real SCA traces containing thousands of sample points. Experimental validation is performed on software-based ASCADv1 and hardware-based AES_HD_ext datasets. In addition, TT-DCNN is shown to be able to learn the exact countermeasure in a best-case setting.

**Keywords:** Side-channel · Neural Network · Deep Learning · Profiling attack · Interpretability · SAT

## 1 Introduction

The increased usage of Internet-of-Things (IoT) devices [Lue21] has led to many applications where the data of the devices manipulated are sensitive, and such devices might be placed in a hostile environment leading to the need to evaluate their security capabilities. Side-channel analysis (SCA) is one of those crucial threats that is required to be evaluated; ever since its first appearance in 1999 [KJJ99], it has become a widely studied research area in cryptography. Physical properties such as timing delay [Koc96], power consumption [KJJ99], and electromagnetic emanation [AARR03] may reveal information on the secret data. SCA focuses on finding and exploiting these leakages to retrieve the secret data. Over the years, the field of SCA has evolved from classical techniques like template attacks [CRR03] to using machine learning algorithms [BLR12, HZ12, HGDM+11, LBM14, LPB+15, GHO15, LBM15, HZ12], and recently deep learning-based SCA [PPM+21].

Received: 2022-10-15 Accepted: 2022-12-15 Published: 2023-03-06

Due to the ever-increasing computing power over the last decades, deep neural networks (DNNs) have gained much recognition in various fields like image recognition [HZRS16], and natural language processing [YHPC17]. In 2016, Maghrebi *et al.* [MPP16] succeeded in retrieving the secret key of an unprotected AES implementation using various types of DNNs, which has drawn much attention over the past few years [PPM$^+$21]. One of the advantages of deep learning-based SCA is that in the presence of countermeasures like masking [PR13] or hiding, it requires little to no preprocessing to obtain a successful attack. This is unlike other classical techniques, which require a tremendous amount of preprocessing before the attack can be mounted [BPS$^+$20, CDP17]. After understanding what DNNs have to offer in SCA, one may want to interpret the black-box algorithm and understand what these DNNs learn from the side-channel traces. Currently, if an attack is unsuccessful, we have no idea why it is so. Furthermore, it is difficult to pinpoint whether the unsuccessful attack is due to the countermeasures or because the hyperparameters and architectures are not correctly tuned for the traces obtained. On the other hand, if the attack is successful, evaluators would not be able to locate the origin of the leakages. Despite that, the designer would want to improve the security of the device. Thus, there is a need for better explainability and interpretability of these DNNs.

In order for us to understand what the DNNs learn, we have to understand where the leakages of the secret information are found on the traces. Masking [PR13] is a very common countermeasure used against SCA. It is theoretically proven to be secure up to a given level against SCA [PR13]. It operates by splitting the secret information into multiple shares. Assuming sequential executions, each share leaks in a different part of the trace. These sample points on the trace that leak are better known as Points of Interest (PoIs). The adversary would need to observe PoIs from all the shares and combine them to retrieve the secret information. As of today, there has not been any work where exact formulae are extracted from the DNN to show the neural network using the PoIs from different shares for key recovery on both seen and unseen traces (i.e., global interpretation).

**Prior Works.** Various works have tried to explore the interpretability of DNN in their own ways. The first work incorporating SCA and DNN interpretability uses Gradient Visualization (GV) [MDP19], which is used in other studies like in [Tim19]. It calculates the gradient of the output of the DNN with respect to the input data. The idea is to observe how a slight change in the sample points in the traces affects the DNN's prediction. The authors of [MDP19] observed that the PoIs obtained through GV for unprotected traces are very similar to the PoIs found by the classical PoI technique like Signal-to-Noise Ratio (SNR). However, they further considered traces with masking order 1 and found that the GV can locate the two PoIs if the DNN is trained with early stopping. In other words, the GV can only pinpoint the PoIs of higher-order masking if the trained DNN did not overfit the dataset. Furthermore, GV also assumes that the neural network is differentiable over the set of all traces (i.e., non-exact formulation), which may not be the case. Hettwer *et al.* extended this work by comparing other explainability techniques, for example Layer-wise Relevance Propagation (LRP) [BBM$^+$15] and Occlusion [ZF13], and visualized them on heatmaps [HGG19]. [GBR22] further compared other techniques like Integrated Gradient [STY17] and SmoothGrad [STK$^+$17] with classical techniques in selecting the PoIs like Difference of Mean (DOM) [KJJ99] or Correlation Power Analysis (CPA) [BCO04]. Although all these techniques are powerful and essential, it only gives an interpretation based on the traces given (i.e., local interpretation). Zaid *et al.* [ZBHV19] provided explainability using weight visualization and feature maps. The feature maps average the values from the same timestamp for the output of a convolution layer, while the weights visualization averages the weights on an MLP layer. The weight visualization proposed by them provides a global interpretation of the weights in the MLPs, while the feature map give a local interpretation. However, both their methods are non-exact formulations of what the DNN learns. In [vdVPB21], Van der Valk *et al.* focus on a different aspect of explainability. They use Singular Vector Canonical Correlation Analysis (SVCCA) on two DNNs with the same architecture but trained on different datasets to compare how correlated the weights of the same layers are. Interestingly, a DNN trained on a dataset in SCA is more correlated to another DNN trained on a dataset in image recognition than to another dataset in SCA with a different countermeasure. However, their work requires a significant computational effort to calculate the correlation between convolution layers [GBC16] and therefore, they only compared the fully connected layers [GBC16] of the DNN. Another technique called ablation was explored to gain insights and better understand the trained DNN in [WWJ+21]. Ablation proceeds by randomly removing some weights or channels in a particular layer in the DNN. The authors then proceed to test the effect of various types of hiding countermeasure on the ablated DNN. They concluded that simpler countermeasures like adding Gaussian noise to the traces are processed in the early layers, while more complex countermeasures like desynchronization are processed in the deeper layers. In terms of interpreting the training of a DNN in SCA, Perin *et al.* [PBP20] created a metric for determining when to stop the training phase based on the work by Shwartz-Ziv, and Tishby [ST17]. They use the Information Bottleneck theory [TPB00] concepts to visualize and interpret the information that the DNN is learning. [PWP22] furthers the work to visualize how shares are processed within each layer of DNN. Recently, [ZBC+22] combined deep learning with a stochastic attack using an autoencoder. Instead of learning the usual discriminative model for key recovery (which easily retrieves the keys), their model learns a generative model. They show that the weights of the neural networks give an equation of the traces corresponding to the leakage. However, for higher-order implementation, the generative model they proposed requires the usage of a classical recombination technique [PRB09], which will increase the time complexity and the length of the traces to analyze. These suggest that analyzing a discriminating model's internal structure remains an open question. Although many works stated above have tried to interpret the DNNs, there are still gaps in understanding them, especially for analyzing traces that have not been seen and providing exact formulae for what they learn.

**Our Contributions.** Our work tries to bridge the gap between explainability and interpretability from SAT equations by using the interpretable neural network called the Truth Table Deep Convolution Neural Networks (TT-DCNNs) [BPKY22]. The TT-DCNN can be used as a discriminative model and provides a transparent inner structure by converting part of the network into SAT equations. The SAT equations offer a representation of the TT-DCNN for us to interpret on both seen and unseen traces, providing us with an exact and global interpretation of the TT-DCNN. To the best of our knowledge, this is the first work interpreting neural networks using SAT equations in the context of side-channel analysis.

The contributions of the paper can be summarized as follows:

1. We provide a general methodology to analyze the SAT equations that are extracted from TT-DCNN in the SCA context.

2. We propose a TT-DCNN-based architecture, which we call $TTSCA_{small}$ and show that $TTSCA_{small}$ can learn the exact locations of the PoIs in simulated traces of different masking orders (i.e., masking order 0 to 3).

3. $TTSCA_{small}$ cannot be directly applied to real traces with hundreds to thousands of sample points. We propose a method to adapt TT-DCNN to overcome the computational limitation due to the patch size (the task of simplifying the SAT equations relies on an NP-complete problem) and, thus, can be used on traces with extended length. We call this adapted architecture $TTSCA_{big}$. We test this architecture on

software-implemented traces, ASCADv1_f (fixed keys) and ASCADv1_r (random keys), and hardware-implemented with low SNR traces, AES_HD_ext.

4. Our analysis shows that our proposed TT-DCNN-based architecture $TTSCA_{big}$ finds the positions of the leakages and learns a function based on these leakages to retrieve the key on both ASCADv1 and AES_HD_ext. The exact formula extracted from our proposed TT-DCNN gives us a global interpretation of what the network learns. In the best case, modified $TTSCA_{big}$ is able to learn the exact masking countermeasure through SAT equations.

In this work, we are only targeting synchronized traces. We validate our approach on higher-order masking in a simulated setting, while validation on real traces is only limited to first-order masking. The results can be publicly accessed on the following weblinks[1,2].

**Paper Organization.** The paper is organized as follows. Section 2 will provide the necessary background on side-channel analysis, deep learning, and TT-DCNN. In Section 3, we give a methodology to analyze the SAT equations acquired from the TT-DCNN with regard to side-channel attacks. In Section 4, we present the $TTSCA_{small}$ and $TTSCA_{big}$ together with the datasets that are tested on. Subsequently, we present the results and interpretability of $TTSCA_{small}$ and $TTSCA_{big}$ on each dataset that was applied and discuss their limitation. Lastly, in Section 5, we conclude the paper and outline some future works.

# 2 Background

## 2.1 Notation and Terminology

We denote sets through the use of calligraphic letters $\mathcal{X}$. The corresponding capital letter $X$ defines a random variable, and the bold capital letter $\boldsymbol{X}$ denotes a random vector. We denote the corresponding lowercase letters $x$ and $\boldsymbol{x}$ to represent the realizations of $X$ and $\boldsymbol{X}$, respectively. We let $\boldsymbol{x}[i]$ stands for the $i^{th}$ entry of a vector $\boldsymbol{x}$. A side-channel trace is defined as a vector $\boldsymbol{t} \in \mathbb{R}^D$ where $D$ is the number of sample points in a trace. Let $C$ represents a cryptographic primitive with $P$ denoting some public variable (e.g., plaintext or ciphertext), and $K$ representing a part of the key. The targeted sensitive variable is the output of the cryptographic primitive, $Z = C(P, K)$ with $Z$ taking values in $\mathcal{Z} = \{s_1, s_2, \dots, s_{|\mathcal{Z}|}\}$. We denote $k$ as the key byte candidate taking its value from the keyspace $\mathcal{K}$ and the correct key as $k^*$.

Masking is a countermeasure that was proven to be secure against side-channel up to a given level of security [PR13]. It splits the targeted sensitive variable $Z$ into many shares. Formally, we say that the cryptographic primitive is of masking order $d$ if $Z$ is split into $d+1$ different shares,

$$Z = \gamma(m_1, m_2, \dots, m_{d+1})$$

such that $Z$ can be obtained back by a generic operator $g$ (e.g., for Boolean masking, $\gamma$ is defined as the XOR of the shares $m_i$ for all $i \in \{1, \dots, d+1\}$). Throughout this paper, we shall focus on the SAT equations representation known as Disjunctive Normal Form (DNF). This is because it provides the most intuitive interpretation of the filters in the neural network, which will be explained in Section 2.3. A DNF, denoted as $dnf$, is defined as a set of Boolean variables $x_1, x_2, \dots, x_h$ such that $dnf = \delta_1 \vee \delta_2 \vee \dots \vee \delta_m$ where each disjunct $\delta_i$ is a conjunction of some literals $\delta_i = l_{i,1} \wedge l_{i,2} \wedge \dots \wedge l_{i,r}$ where each $l_{i,j} \in \{x_1, x_2, \dots, x_h\}$.

[1] For $TTSCA_{small}$: https://github.com/yap231995/TTSCA_small

[2] For $TTSCA_{big}$: https://github.com/yap231995/TTSCA_big

## 2.2 Profiling Attacks

Profiling attacks assume a worst-case scenario where the adversary has access to two similar devices: a prototype or clone device and a target device. For the prototype device, the adversary can manipulate or know the device's key while the key for the target device is unknown to him. Furthermore, the adversary is able to collect several traces from a known set of random plaintexts (or ciphertexts) from both devices. The adversary's goal is to break and retrieve the unknown key from the target device.

Profiling attacks can be divided into two stages: the profiling phase and the attack phase. In the profiling phase, the adversary will build a distinguisher $\mathcal{F}$ that takes in a set of profiling traces from the prototype device and returns a conditional probability mass function $\Pr(\boldsymbol{T}|Z = z)$. During the attack phase, the distinguisher outputs a probability score for each hypothetical sensitive value $\boldsymbol{y}_i = \mathcal{F}(\boldsymbol{t}_i)$ for each attack traces $\boldsymbol{t}_i$ acquired from the target device. For every key $k \in \mathcal{K}$, the log likelihood score is defined as:

$$s_{N_a}(k) = \sum_{i=1}^{N_a} log(\boldsymbol{y}_i[z_{i,k}])$$

where $N_a$ as the number of attack traces used and $z_{i,k} = C(p_i, k)$ are the hypothetical sensitive values based on the key $k$ with $p_i$ being the corresponding public variable to the trace $\boldsymbol{t}_i$. The adversary or evaluator can rank the key of the log-likelihood score in a decreasing order and classify them into a guess vector $\boldsymbol{G} = [G_0, G_1, \ldots, G_{|\mathcal{K}|-1}]$. The key corresponds to the score $G_0$ is the most likely candidate, and the key of the score $G_{|\mathcal{K}|-1}$ is the least likely candidate. The index of guess vector $\boldsymbol{G}$ is called the rank of the key. The metric called guessing entropy $GE$ is defined as the average rank of the correct key $k^*$ [SMY09]. If $GE = 0$, when using $N_a$ attack traces, the attack is considered successful.

In deep learning-based SCA, we train a DNN, $f_{\boldsymbol{\theta}}$, as the distinguisher where $\mathcal{F} = f_{\boldsymbol{\theta}}$. The most commonly found DNNs used in SCA are the Multilayer Perceptrons (MLPs) and Convolutional Neural Network (CNN), but they are not interpretable. Therefore, in the next section, we will describe an interpretable DNN that we will be exploring.

## 2.3 Truth Table Deep Convolutional Neural Network

In this section, we present the interpretable neural network called Truth Table Deep Convolutional Neural Networks (TT-DCNNs) proposed by Benamira *et al.* [BPKY22], which is convertible into SAT formulas by design. This is achieved by first lowering the number of connections from one convolution layer to another and also by using the Heaviside step function [Bra78], denoted as $bin_{act}$, to binarize the features while still having real-valued weights. In order to train without much loss in performance when using the Heaviside step function, Benamira *et al.* adopted the Straight-Through Estimator (STE) proposed by [HCS+16]. Since the traces are one-dimensional, unlike the images, which are two-dimensional, we apply the 1D-convolutional layers in our TT-DCNN instead. For 2D-convolutional TT-DCNN, we refer readers to the original paper [BPKY22]. In the next few paragraphs, we shall construct the TT-DCNN architecture step by step for 1D-CNN.

**1D-convolution, one input channel.** Suppose we denote $\Phi_F$ to be a 1D-convolution associated with the filter $F$ of kernel size $ker = n$, stride $s$ and no padding, and let the input feature with a single input channel $chn_{input} = 1$ be represented as $v_0 \ldots v_{N-1}$ where $N$ is the length of the input feature. We define $y_{i,F}$, the output of the function $\Phi_F$ at position $i$, and $y_{bin,i,F}$ as:

$$y_{bin,i,F} = bin_{act}(y_{i,F}) = bin_{act}(\Phi_F(v_{i\times s}, v_{i\times s+1}, \ldots, v_{i\times s+(n-1)})).$$

If $v_{i\times s+q}$ are binary values (i.e., $\{0,1\}$) for all $q = 0,\dots,n-1$, we can express the 1D-convolutional layer $\Phi_F$ as a truth table by enumerating all $2^n$ possible input combinations. The truth table can then be converted into a simplified DNF formula using the Quine–McCluskey algorithm [Bla38] for interpretation. However, there is a computational limitation when using this algorithm because the algorithm is solving an NP-complete problem [UVSV06]. Therefore, we shall limit ourselves to $n \leq 12$ for this paper.

**Example 1.** We consider a 1D-convolution with one filter of kernel size $ker = 4$, a stride of size 2, and no padding. Let the weights of the 1D-convolutional layer be $\boldsymbol{W_1} = (10,-1,3,-5)$. We define the following DNF literals $[x_0 \quad x_1 \quad x_2 \quad x_3]$. As the inputs and output are binary and the number of input entries for the CNN layer is 4 (kernel size = 4 and number of input channel $chn_{input} = 1$), we have $2^4 = 16$ possible entries: $[0 \quad 0 \quad 0 \quad 0]$, $[0 \quad 0 \quad 0 \quad 1]$, $\cdots$, $[1 \quad 1 \quad 1 \quad 1]$. For each input, we calculate the corresponding output resulting from the convolution of $\boldsymbol{W_1}$ with the 16 possible literal entries: $[0,-5,3,-2,-1,-5,3,-2,10,5,13,8,9,4,12,7]$. After binarization with the Heaviside step function (i.e., $Bin(x) = \frac{1}{2} + \frac{sgn(x)}{2}$), we have $y_{binary} = [0,0,1,0,0,0,1,0,1,1,1,1,1,1,1,1]$. We then apply the Quine-McCluskey algorithm [Bla38] which gives $dnf = (x_2 \wedge \ x_3) \vee x_0$ (a simplified DNF). This procedure provides us with the general form of $bin_{act} \circ \Phi_F(\cdot)$ with binary inputs.

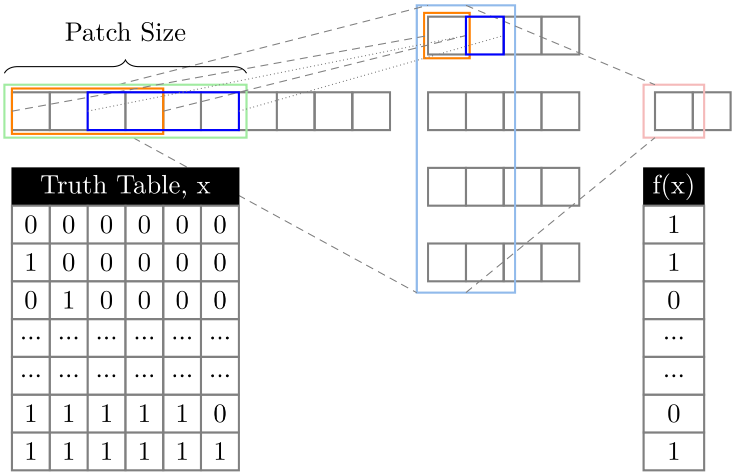
**1D-convolution, multiple input channels.** In general, a 1D-convolution takes in several channels as input (i.e., $chn_{input} > 1$), and therefore the number of input variables of $\Phi_F$ will rises substantially. For example, a 1D-CNN that takes 32 input channels with a kernel size of 4 yields an input of size 128, which well exceeds our limit of $n = 12$. In order to overcome this, we group the channels using the group parameter [DV16]. Grouped convolutions divide the input channels into $g$ groups, then apply separate convolutions within each group; this effectively decreases the number of inputs of $\Phi_F$ to each individual filter by a factor of $g$. In that case, we have $n = ker * chn_{input}/g$ where $chn_{input}$ is the number of input channels, and $ker$ is the kernel size. We remark that the number of output channels must be a multiple of the number of input channels.

**Example 2.** We consider a 1D-convolutional layer with one filter of kernel size $ker = 4$, a stride of size $s = 2$, and no padding. Let the 1D-CNN weights be $\boldsymbol{W_1} = \begin{bmatrix} 10 & -1 & 3 & -5 \\ 8 & -2 & 5 & -1 \\ -3 & 0 & 2 & 4 \\ -10 & 1 & -3 & 5 \end{bmatrix}$ for the 4-channel input. We can observe that $\boldsymbol{W_1}$ requires $16 = 4 \times 4$ binary inputs, which leads to a truth table of size $2^{16}$, which is too large according to our previous criteria fixed at $2^{12}$. Therefore, by defining a group $g = 2$, then $W_1$ becomes 2 filters matrices: $\boldsymbol{W_{1,1}} = \begin{bmatrix} 10 & -1 & 3 & -5 \\ 8 & -2 & 5 & -1 \end{bmatrix}$ for filter 1 which takes as input only the first two input channels, and $\boldsymbol{W_{1,2}} = \begin{bmatrix} -3 & 0 & 2 & 4 \\ -10 & 1 & -3 & 5 \end{bmatrix}$ for filter 2 which takes as input only the last two input channels. Therefore, thanks to this grouping, each filter has an input size of $n = 8 = 4 \times 4/2$ which fits our criteria of $n \leq 12$.

**DNN $\Phi_F$ function, multiple input channels.** In general, if $\Phi_F$ is a 1D-convolution, the TT-DCNN will not be able to learn complex tasks such as image classification [BPKY22]. Therefore, we need to transform the linear function $\Phi_F$ into a non-linear one to increase the learning capacity of the neural network. The idea is to use a DNN architecture for $\Phi_F$ without increasing the input size of $\Phi_F$. The $\Phi_F$ input size is now defined as $n = pc/g$ where $p$ is the patch size[3] instead of the kernel size $ker$. Figure 1 illustrates an example of a

[3] The patch is the region of the input that produces the feature, which is commonly referred to as the receptive field [ANS19].

$\Phi_F$, with one input channel and $g = 1$ with a patch size of 6. $F$ now refers to the $F^{th}$ output channel of the last layer for DNN $\Phi_F$. In [BPKY22], the authors proposed an architecture for $\Phi_F$ so-called Learning Truth Table (LTT) block. The LTT block is built upon a layer known as the amplification layer. The amplification layer works by simply adding a new convolution layer with kernel size 1 after a convolution layer. Doing so will not increase the patch size. Overall, the LTT block comprises two 1D-convolution layers, denoted as Conv1D, with an amplification parameter $\tau$, which is the ratio between the number of channels of the first layer and the number of channels of the second layer. Each layer in the LTT block is followed by a batch normalization [IS15] and a non-linear activation function. In our case, for the first non-linear activation function, we are using SeLU [KUMH17] and the second non-linear activation is $bin_{act}$. Figure 2 shows the internal working and an overview of the LTT block. We will define our $\Phi_F$ for our proposed architectures $TTSCA_{small}$ and $TTSCA_{big}$ in Section 4.1 and Section 4.2 respectively.

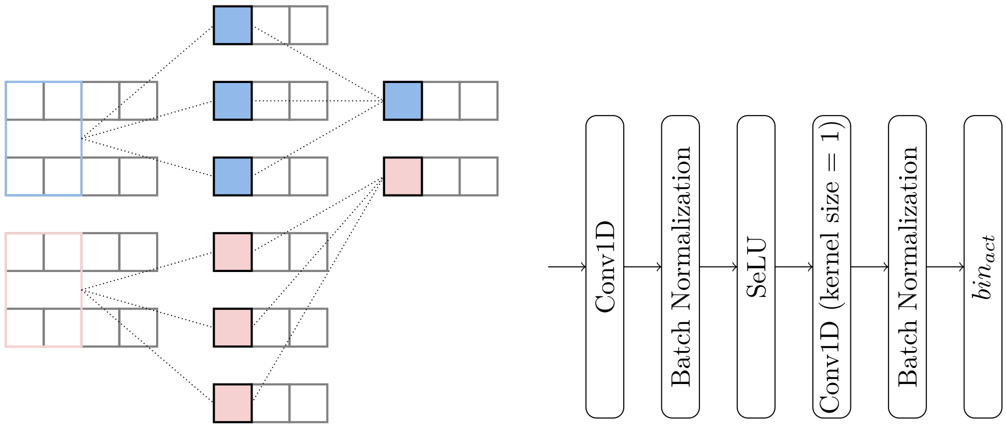


| Truth Table, x | | | | | | f(x) |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| ... | ... | ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... | ... | ... |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**Figure 1:** Example of converting a $\Phi_F$ into a truth table. The above example has two layers. The first layer has an input channel = 1, output channel = 4, kernel size = 4 with stride = 2, and a second layer of input channel = 4, output channel = 1, kernel size = 2 with stride = 2. The patch size $p$ of $\Phi_F$ is 6 (i.e., green box) since the output feature (i.e., pink box) requires 6 input entries (i.e., orange and blue box).

**Why TT-DCNN in SCA?** There are other works that use SAT equations for their neural network, namely the Binarized Neural Network (BNN) [HCS+16], the Concept Rule Sets (CRS) [WZLW19], and the Rule-based Representation Learner (RRL) [WZLW21]. The BNN and TT-DCNN are the only CNN-based networks that use SAT equations, while the CRS and RRL are MLP-based networks that comprise of SAT formulae. Nonetheless, we want to analyze CNN-based DNNs that found great success among the other DNNs [MPP16] due to their shift-invariant nature. Therefore, the only two CNN-based DNNs that use SAT equations are BNN and TT-DCNN. However, a BNN loses its interpretability when its binarized convolution block is converted into an inequality for pseudo-Boolean constraint [RM21, CNHR18, JR20], which is consequently mapped into SAT formulae. Moreover, the SAT formulae of a BNN contains a large amount of disjuncts/clauses compared to TT-DCNN, which is intractable to analyze. Furthermore, the TT-DCNN is more expressive compared to BNN because TT-DCNN contains real-valued weights, unlike BNN, which only has binarized weights. Therefore, TT-DCNN is a preferred choice.

Although the original design for TT-DCNN is meant for the adversarial attack on image datasets [BPKY22], TT-DCNNs transparent inner structure allows us to interpret the neural network easily by describing the rules/decisions made by the neural network through its DNF equations. Therefore, we can use them in SCA context to pinpoint the location of the PoIs used. Using the DNF here, tell us which sample points in the traces are used by the network in retrieving the secret key. Based on our proposed TT-DCNN-based architectures stated in Section 4, one sample point of the traces or a window of sample points corresponds to one literal. Moreover, the AND operations $\wedge$ in a disjunct show which literals are to be jointly used together. These disjuncts will give us the position of the leakage points that are exploited by our TT-DCNN-based neural network. The disjuncts of the DNF, which are the exact and interpretable formulation of the TT-DCNN-based neural network, also provide us with a global interpretation by giving us the decision of the network even on traces that the network has not encountered, unlike the usual CNN or BNN.



**(a)** Learning Truth Table Block (LTT) internal working: first Conv1D layer of kernel size 2, stride 1 and group $g = 2$. The amplification layer (second Conv1D layer) has an amplification parameter, $\tau = 3$ of kernel size 1 and stride 1.

**(b)** Learning Truth Table Block (LTT) overview: the Conv1D with kernel size = 1 is the amplification layer.

**Figure 2:** Learning Truth Table Block (LTT)

## 3 Methodology of Analyzing DNF Equations for Trained TT-DCNN in SCA

After converting the filters in TT-DCNN into DNF while using the Quine-McCluskey algorithm to simplify the DNFs, we found that there are many disjuncts or literals that are unnecessary in retrieving the secret key. This is likely because the neural network overfits [BPKY22] or underfits the dataset. Therefore, we propose three different steps to remove these unnecessary disjuncts or literals to find the most miniature set of rules that the neural network needs for key recovery:

1. Sieving disjuncts based on their size,
2. Separating disjuncts based on their combinations of literals (CoLs),
3. Trimming disjuncts based on the literals.

We remark that the three steps stated are heuristic approaches, and finding all sets of rules or optimal rules remains an open problem. In addition, after each step, we set the channel as 0 when it does not have any disjuncts left.

**Sieving Disjuncts Based on Their Size.** As described by [BPKY22], a large number of literals in a disjunct is possibly due to overfitting. On the other hand, we found that some disjuncts with a small number of literals are irrelevant in retrieving the keys, which probably corresponds to the neural network underfitting the dataset. Therefore, we define the size of the disjunct as the number of literals it contains and sieves the disjuncts based on their size. We replace the original disjuncts of TT-DCNN with the disjuncts of a given size and compute its guessing entropy. We repeat this process individually for each disjunct size. Then we compare the guessing entropy of each disjunct's size and search for the minimum size such that $GE = 0$ (this criterion may be adapted for other use cases). Although there might be another set of disjuncts within the other disjuncts sizes used to retrieve the key, considering all the disjuncts' sizes that give us $GE = 0$ will become too intractable. Moreover, taking into account the smallest disjuncts' size allows us to observe the least number of literals and, therefore, the fewest number of sample points (or window of sample points) needed by the TT-DCNN to retrieve the secret key successfully.

**Separating Disjuncts Based on Their Combinations of Literals (CoLs).** After separating the disjuncts based on their sizes and obtaining the minimum disjuncts' size with a $GE = 0$, there might remain a considerable amount of disjuncts to interpret (For ex., 198 disjuncts of size 4 for ASCADv1_f dataset and 190 disjuncts of size 3 for AES_HD_ext). Therefore, we would like to separate them further based on the disjuncts' combinations of literals (CoLs) which we define below.

**Definition 1.** The **Combinations of Literals (CoL)** for $(x_{i_0}, x_{i_1}, \ldots, x_{i_{a-1}})$ is denoted as the set of disjuncts

$$\{(w_1 \wedge w_2 \wedge \cdots \wedge w_a) \mid w_h \in \{x_{i_h}, \neg x_{i_h}\} \text{ for } h = 0, \ldots, a-1\}.$$

For example, $(x_1 \wedge \neg x_3 \wedge x_5 \wedge \neg x_6)$ and $(\neg x_1 \wedge x_3 \wedge x_5 \wedge x_6)$ are disjuncts that are contained in the CoL for $(x_1, x_3, x_5, x_6)$.

After separating the disjuncts by size, we generate the list of unique CoLs from the disjuncts with the minimum size and denote it as $Lst_{unique}$. Among the different unique CoLs, we want to obtain a set of CoLs such that replacing the original set of disjuncts acquired from the trained TT-DCNN with it can still successfully recover the secret key. We call these CoLs crucial for key recovery as critical CoLs. If the number of unique CoLs is small ($\leq 5$), we can check all their combinations. However, if the trained TT-DCNN has many CoLs, we would require a viable approach to find the critical CoLs. Firstly, we replace the original disjuncts of TT-DCNN with disjuncts of a given CoL and compute its guessing entropy. We repeat this process for each CoL found in $Lst_{unique}$ independently. This is because sometimes, the trained TT-DCNN only requires one CoL to retrieve the key successfully. However, some trained neural networks may require more than one CoL for key recovery; therefore, we propose an algorithm to acquire the critical CoLs for further analysis (see Algorithm 1).

The main idea of Algorithm 1 is first to set $Lst_{in}$ as the list of all unique CoLs, $Lst_{unique}$, then remove one CoL from $Lst_{in}$ and check if the guessing entropy of the correct key increases above a certain threshold $\lambda$. Throughout the paper, we set $\lambda = 1$. If the guessing entropy increases above the threshold $\lambda$, it means that the removed CoL is crucial for recovering the key, so we put it back into $Lst_{in}$. On the contrary, if the guessing entropy did not increase above the threshold $\lambda$, we can remove the CoL from $Lst_{in}$ and place them into $Lst_{out}$, as they are currently not required for retrieving the secret key. Note that the algorithm only helps to find one set of critical CoLs. Algorithm 1 relies on the order of the list $Lst_{unique}$, and therefore, the existence of another set of critical CoLs is possible.

**Algorithm 1** CriticalCoLs

```
Input:
DNFs of all the filter $DNF_{start}$,
List of unique CoLs of $DNF_{start}$ denote as $Lst_{unique}$.
Threshold, $\lambda$
Output:
List of critical CoLs $Lst_{in}$,
procedure CriticalCoLs($L_{unique}$)
    $Lst_{in} = Lst_{unique}$
    $DNF_{in} = DNF_{start}$
    $Lst_{out} = \{\}$
    for CoL $comb_{lit}$ in $L_{unique}$ do
        Remove $comb_{lit}$ from $Lst_{in}$
        Remove disjuncts contain in $comb_{lit}$ from $DNF_{in}$
        Run guessing entropy metric of the correct key $k^*$, $GE_{k^*}$ with using $DNF_{in}$ as part of the trained TT-DCNN
        if $GE_{k^*} \geq \lambda$ then
            Add $comb_{lit}$ back into $Lst_{in}$
            Add back the removed disjuncts contain in $comb_{lit}$ to $DNF_{in}$
        else
            Add $comb_{lit}$ into $Lst_{out}$
        end if
    end for
    return $Lst_{in}$
end procedure
```

**Trimming Disjuncts Based on the Literals.** Some of the disjuncts might contain literals with information that is not useful. We introduce an operation called trimming. We trim a literal ($x_i$) from a disjunct $\delta$ simply by removing $x_i$ if the literal appears in the disjunct $\delta$ regardless of whether it is the negation. For example, if we trim the literal ($x_2$) from ($x_1 \wedge \neg x_2 \wedge x_4 \wedge \neg x_6$) and ($\neg x_1 \wedge x_2 \wedge x_4 \wedge x_6$) then we will obtain ($x_1 \wedge x_4 \wedge \neg x_6$) and ($\neg x_1 \wedge x_4 \wedge x_6$) respectively.

When the number of literals found is small, it is possible to exhaust all possible scenarios for trimming and find the set of literals that has the highest impact on the guessing entropy. However, the maximum number of possible cases is $2^n$ (i.e., when every single literal is found), which could be too computationally intensive. This is not to mention that the calculation of the guessing entropy is also very expensive. As a result, we want to trim out the literals that have minimal impact on the guessing entropy (i.e., literals that will lead to key recovery) without enumerating all the possible cases. Therefore, we propose the following algorithm called individual trimming algorithm. Given a set of disjuncts $\mathcal{S}$, we trim $x_i$ from $\mathcal{S}$ and check its guessing entropy, then repeat this for all $i = 0, \ldots, n-1$. We set these algorithms after the previous two steps as we do not want to miss out on crucial literals required for key recovery (see Appendix A.2 for examples).

# 4 Experimental Results

In this section, we propose two TT-DCNN-based neural networks called $TTSCA_{small}$ and $TTSCA_{big}$. We train $TTSCA_{small}$ on a small-sized simulated dataset and $TTSCA_{big}$ on real-measurement traces ASCADv1 and AES_HD_ext. We also analyze their acquired SAT equations with the proposed methodology from Section 3.

## 4.1 The TT-DCNN-based Neural Network, $TTSCA_{small}$

The TT-DCNN-based architecture $TTSCA_{small}$ is illustrated in Figure 3. The $TTSCA_{small}$ apply a batch normalization layer on the traces followed by a Heaviside step function $bin_{act}$. Three layers of 1D-convolution layers are applied thereafter, with each 1D-convolution layer utilizing a convolution operation, then a SeLU activation function, and lastly, a batch normalization. The parameters of each 1D-convolution layer can be found in Table 1, and the patch size for the $TTSCA_{small}$ is 9, equal to the kernel size of the filter in the Conv1D 2. Moreover, instead of using MLP layers after the Flatten operation, we use a linear regression layer to make the neural network fully interpretable.

We notice that using $bin_{act}$ before the Flatten operation only after training and applying a $bin_{act}$ after the Conv1D layer 2 are necessary in recovering the key. If we apply the $bin_{act}$ before the Flatten operation during training or remove the $bin_{act}$ after the Conv1D layer 2, we observe that it does not successfully recover the secret key. This is possibly due to the simplicity of the dataset, where some loss of information after the Conv1D layer 2 is required, but training with a $bin_{act}$ before the Flatten operation will result in too much information loss. Furthermore, the application of $bin_{act}$ before the Flatten operation only after training is required to convert the three 1D-convolution layers into truth tables. Therefore, the function $\Phi_F$ in $TTSCA_{small}$ consists of all the Conv1D layers and the $bin_{act}$ after the Conv1D layer 2 (colored green in Figure 3).

We employ the Glorot weight initialization [GB10], the One Cycle Policy [ST18] with learning rate of 0.0025 to 0.005, a $L_2$ norm with regularization factor of 0.00125 and the Adam optimizer [KB17] when training $TTSCA_{small}$. We train $TTSCA_{small}$ over 9 epochs.

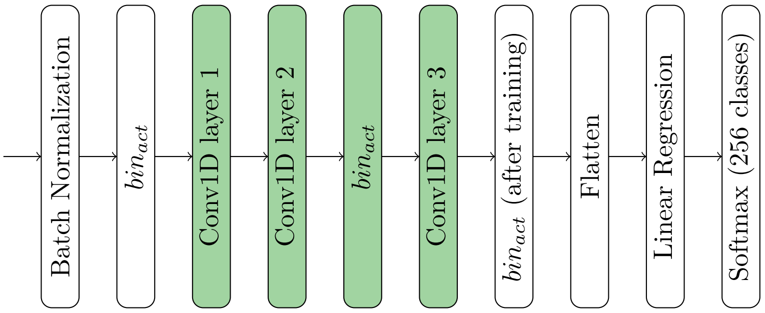


**Figure 3:** Overview of $TTSCA_{small}$ architecture.

**Table 1:** Parameters of each layers in $TTSCA_{small}$.

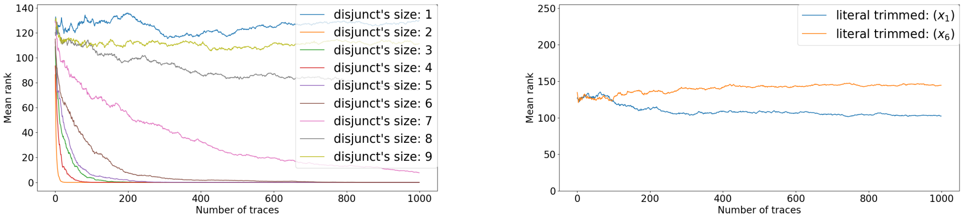
| **Conv1D** | **Parameters** |
|---|---|
| 1 | Number of filters/channels = 16,kernel size = 1,stride = 1. |
| 2 | Number of filters/channels = 16, kernel size = 9, stride = 4. |
| 3 | Number of filters/channels = 16, kernel size = 1, strides = 1. |

### 4.1.1 Simulated Data

We generated our own simulated dataset using Python code similar to [Tim19]. For each trace that was generated, it has 20 sample points. We denote each trace as an array $trace[0\dots19]$ and set $d+1$ number as the leakage point where $d \in \{0, 1, 2, 3\}$ is the masking order of the generated dataset. The remaining $20-(d+1)$ sample points are randomly generated bytes. Then we add a noise based on the Gaussian distribution with a mean of 0 and a standard deviation of 0.1 to all sample points. We denote $Z := SBox(pt \oplus key)$ to be the sensitive variable, where $pt \in \{0, \dots, 255\}$ being the plaintext byte, $SBox$ is the substitution box of the Advanced Encryption Scheme (AES) [DBN+01], and $key = 0x03$ is the fixed key byte. We generate the traces according to the Table 2 where $m_1, m_2$ and $m_3$ are randomly generated bytes. We use 14000 traces for the profiling phase and 5000

**Table 2:** Leakage points of the traces generated in simulated data based on the masking order.

| Masking Order $d$ | Leakage Point |
|---|---|
| 0 | $trace[10] = Z$ |
| 1 | $trace[10] = Z \oplus m_1$<br>$trace[5] = m_1$ |
| 2 | $trace[10] = Z \oplus m_1 \oplus m_2$<br>$trace[5] = m_1, trace[8] = m_2$ |
| 3 | $trace[10] = Z \oplus m_1 \oplus m_2 \oplus m_3$<br>$trace[5] = m_1$ , $trace[8] = m_2$ , $trace[12] = m_3$ |



**(a)** Different sizes of disjuncts.

**(b)** Trimming with $x_1$ and $x_6$.

**Figure 4:** Guessing entropy of $TTSCA_{small}$ for simulated data with masking order 1.

traces for the attack phase.

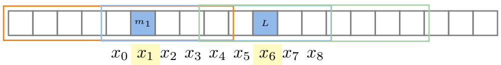
**Interpretation of the DNF Equations of $TTSCA_{small}$ on Simulated Data.** After training $TTSCA_{small}$ on the simulated traces, we convert the three 1D-convolution layers into DNF equations by considering the truth tables obtained through enumerating all $2^n$ inputs. We shall only focus on masking order 1 here, since the results for masking orders 0 to 3 all show that $TTSCA_{small}$ is able to pinpoint the position of the leakage that was allocated in the simulated traces.

We first sieve the disjuncts based on their sizes (see Figure 4a). Figure 4a shows that the disjuncts with size 2 to 5 all have $GE = 0$. We shall focus on the smallest size that has $GE = 0$ as stated in Section 3; in this case, 2 is the smallest size (orange line in Figure 4a). For our trained $TTSCA_{small}$, the disjuncts of size 2 are just $(x_1 \wedge \neg x_6), (x_6 \wedge \neg x_1)$ and $(\neg x_6 \wedge \neg x_1)$. Since the CoL for $(x_1, x_6)$ is the only CoL, we simply try to trim the literals $(x_1)$ and $(x_6)$ individually. We observe from Figure 4 that $GE \neq 0$ when we trim $(x_1)$ or $(x_6)$. This shows that both $x_1$ and $x_6$ are literals that the trained $TTSCA_{small}$ necessary for key recovery. Figure 5b illustrate the neural network's patch as it slides through a trace. In the second timestamp when this patch slides through the trace (i.e. light blue box in Figure 5b), the literals that corresponds to PoIs of the shares $m_1$ and $L = Z \oplus m_1$ are $x_1$ and $x_6$ respectively. Therefore, we can conclude that our $TTSCA_{small}$ has learned a function of $m_1$ and $Z \oplus m_1$ based on $x_1$ and $x_6$ such that it retrieved the secret key $key$. The results for masking orders of $0, 2$ and 3 also show that the $TTSCA_{small}$ learns a function for key recovery based on the literals to which the leakage points correspond. For each masking order $1, 2$ and 3, an example of a disjunct acquired from $TTSCA_{small}$ that is necessary for key recovery is shown in Table 5a. We conclude that $TTSCA_{small}$ can pinpoint the leakage's position and use it to retrieve the key.

Next, we show that $TTSCA_{small}$ is unable to recover the secret key if there exists at any point in time PoIs corresponding to a share that are not within the patch. We show this by training $TTSCA_{small}$ on new simulated traces of masking order 1. We set $m_1$

| Masking Order | Example Of Disjunct |
|---|---|
| 1 | $(x_1 \wedge \neg x_6)$ |
| 2 | $(x_1 \wedge \neg x_4 \wedge \neg x_6)$ |
| 3 | $(x_1 \wedge x_4 \wedge x_6 \wedge x_8)$ |

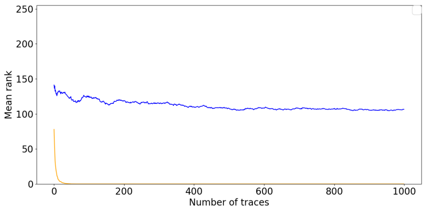
**(a)** Example of disjuncts of $TTSCA_{small}$ retrieved which allow key recovery.



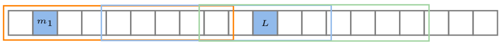
**(b)** Orange, light blue and light green boxes shows that patch size when it is at timestamp $1, 2$ and $3$ respectively. Denote $L = Z \oplus m_1$.

**Figure 5:** Example of disjuncts of $TTSCA_{small}$ necessary for key recovery and a visualization of a simulated trace.

and $L = Z \oplus m_1$ at $trace[1]$ and $trace[10]$ respectively in the new simulated traces (see Figure 6b). Observe that at any point of time, $m_1$ and $L = Z \oplus m_1$ are not within the patch together. We observe that $GE \neq 0$ when $TTSCA_{small}$ is trained on the new simulated traces (see Figure 6a). Therefore, at any point in time, both $m_1$ and $L = Z \oplus m_1$ need to lie within the same patch for $TTSCA_{small}$ to retrieve the key successfully. This solidifies the claim that capturing long-distance dependency in the deep learning literature [SXW+20] is also an issue in the context of SCA.



**(a)** Blue: Guessing entropy of $TTSCA_{small}$ trained on the new simulated traces of masking order 1 when $m_1$ and $L = Z \oplus m_1$ place at $trace[1]$ and $trace[10]$ respectively.
Orange: Guessing entropy $TTSCA_{small}$ trained on simulated traces of masking order 1 stated in Table 2.

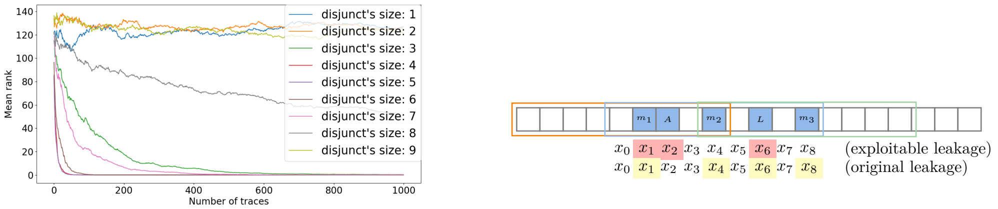


**(b)** Orange, light blue and light green boxes shows that patch size when it is at timestamp $1, 2$ and $3$ respectively. Denote $L = Z \oplus m_1$.

**Figure 6:** Guessing entropy of $TTSCA_{small}$ trained on new simulated traces and a visualization of the new simulated trace with masking order 1.

**Masking with Flaws:** Next, we explore the interpretability of $TTSCA_{small}$ for masked implementations with flaws [EST+22, MGH14]. Flaws in a $d-$order masking scheme can result in exploitable leakage at lower security order $\leq d$. Such flaws can be attributed to several factors like bugs in the implementation, processor pipeline or register overwrites between two shares etc. We create another simulated dataset for $d = 4$ with a reduction in security order to $d - 1$ by introducing a flaw $(m_2 \oplus m_3)$ at $trace[6]$ (see Figure 7b), due to combination of two shares.

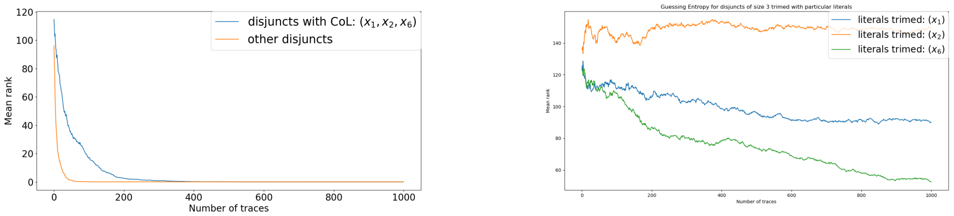
After training $TTSCA_{small}$, we convert the $TTSCA_{small}$ into DNF equations for analysis. As before, we sieve the disjuncts based on their size. The disjuncts with sizes $3, 4$ and $5$ each obtain $GE = 0$ (see Figure 7a). If $TTSCA_{small}$ indeed learns a function based on $x_1, x_2$ and $x_6$, then $TTSCA_{small}$ will learn the exploitable leakage from mask combination (see red literals in Figure 7b). Therefore, we separate the disjuncts with size 3 based on the CoL of $(x_1, x_2, x_6)$ (see Figure 8a). We further check if all these three literals are necessary by using the individual trimming algorithm. From Figure 8b, we obtain $GE \neq 0$ whenever one of the literals is not present. This shows that all three literals are



**(a)** Guessing entropy of each individual size of $TTSCA_{small}$ for flawed masking.

**(b)** Denote $L = Z \oplus m_1$ and $A = m_2 \oplus m_3$.

**Figure 7:** Guessing entropy when using step 1 and a visualization of the simulated data with flawed masking.



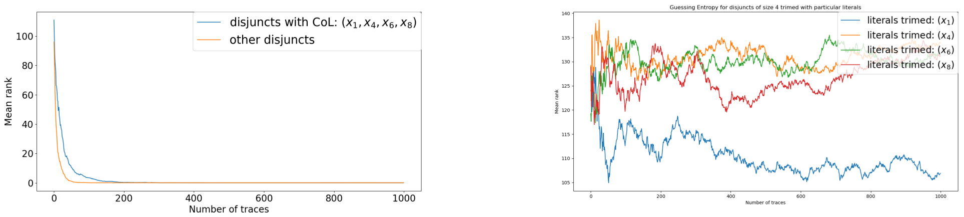
**(a)** CoL of $(x_1, x_2, x_6)$.

**(b)** Trimmed literals to check relevancy of $x_1, x_2$ and $x_6$.

**Figure 8:** Guessing entropy of $TTSCA_{small}$ on simulated traces of masking order 3 with flaws.

important.

Moreover, we check if $TTSCA_{small}$ also learns the original $d$-order leakage simultaneously. If $TTSCA_{small}$ learns a function based on $x_1, x_4, x_6$ and $x_8$ then $TTSCA_{small}$ will learn the $d$-order leakage (see yellow literals in Figure 7b). As a result, we separate the disjuncts of size 4 based on the CoL of $(x_1, x_4, x_6, x_8)$ and obtain $GE = 0$ (see Figure 9a). We further verify if all these four literals are necessary for key recovery. Figure 9b shows that trimming any one of these literals will result in $GE > 0$, which reveals to us that $x_1, x_4, x_6$ and $x_8$ are required for key recovery.



**(a)** CoL of $(x_1, x_4, x_6, x_8)$.

**(b)** Trimmed literals to check relevancy of $x_1, x_4, x_6$ and $x_8$.

**Figure 9:** Guessing entropy of $TTSCA_{small}$ on simulated traces of masking order 3 ($d + 1$-order leakage).

## 4.2 The TT-DCNN Neural Network, $TTSCA_{big}$

As discussed in Section 4.1.1, the $TTSCA_{small}$ is unable to successfully recover the key if there exists a share whose PoIs are not within the patch at any point in time. Meanwhile, there are hundreds to thousands of sample points in real traces, more than our small-sized simulated traces of 20 sample points, and PoIs corresponding to different shares might be hundreds of sample points away from each other. The most obvious way to apply $TTSCA_{small}$ onto longer traces is to increase its patch size but increasing the patch size above 12 will become intractable to compute as the Quine–McCluskey algorithm is solving an NP-complete problem whenever it simplifies a DNF equation. Therefore, applying $TTSCA_{small}$ directly on traces with extended length is impossible. Nonetheless, we would want a TT-DCNN-based neural network with a patch size below or equal to 12 while still able to interpret its DNF equations with respect to the sample points without losing its ability to retrieve the secret key when trained on longer traces. We propose a new TT-DCNN-based architecture called $TTSCA_{big}$, which achieves all of that.

The $TTSCA_{big}$ first consists of a 1D-convolution layer, followed by a batch normalization and subsequently with an average pooling layer. Then, a $bin_{act}$ is applied before the LTT block. The LTT block is meant to be converted into SAT equations. Thereafter, we apply the Flatten operation before utilizing three linear regressions. We use linear regressions instead of MLPs since they are interpretable compared to MLPs. Figure 10 shows $TTSCA_{big}$ architecture.

To overcome the limitation of patch size, the first 1D-convolution layer, batch normalization, and average pooling [GBC16] are considered as a preprocessing block. This preprocessing block converts each window of sample points into one literal allowing the patch size to be contained within the acceptable range. Furthermore, the average pooling is tuned to make the windows of sample points not overlap, which allows for easier interpretation.

Our training methodology is as follows: we apply the horizontal standardization on the traces in the same way as [WAGP20] and employ the He initialization [HZRS15], the One Cycle Policy with a learning rate of 0.005, and the Adam optimizer [KB17] to train $TTSCA_{big}$ over 50 epochs.

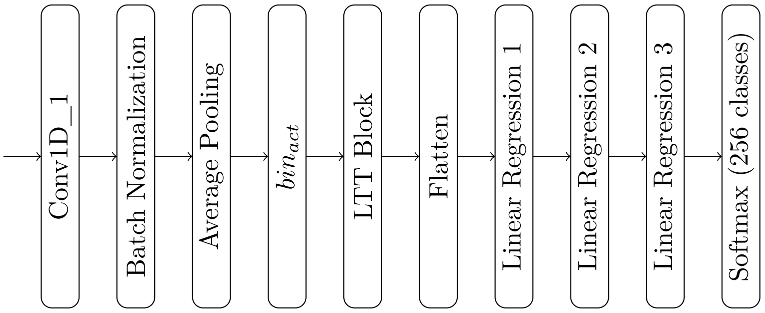


**Figure 10:** $TTSCA_{big}$ architecture.

### 4.2.1 ASCADv1

ASCADv1 is a first-order software AES implementation running over an 8-bit *AVR* architecture on the *ATMega8515* device with an operating frequency of 4 MHz [BPS$^{+}$20]. We focus on the synchronized traces dataset and target the third byte of the first round Sbox, which we shall denote as $SBox(pt \oplus k^{*})$ where $pt$ is the third plaintext byte, and $k^{*}$ is the third byte of the first round secret key. Since it is a first-order implementation, the common output mask of the Sbox is denoted as $r_{out}$. It consists of two forms: fixed keys (denoted as ASCADv1_f) and random keys (denoted as ASCADv1_r). The authors of [BPS+20] pre-selected 700 sample points from raw traces for ASCADv1_f and 1400 sample points for ASCADv1_r, corresponding to the information regarding the third byte of the first round Sbox. ASCADv1_f consists of 60k traces with 50k traces used for profiling and 10k for attacking. On the other hand, ASCADv1_r consists of 300k traces with 200k traces used for profiling and 100k for attacking.

**Interpretation of the DNF Equations of $TTSCA_{big}$ on ASCADv1_f.** We trained a $TTSCA_{big}$ with parameters indicated in Table 3. Since the first 1D-convolution layer, batch normalization, and average pooling layer are considered a preprocessing block, the only portion of $TTSCA_{big}$ where we convert into DNF equations is the LTT block (i.e., $\Phi_F$ is the LTT block). Due to the preprocessing block, we obtain a patch size of 7 where each literal represents a set of sample points in the trace. Table 4 shows which sample points each literal represents, which can be derived from Figure 20 in Appendix A.1.

**Table 3:** Parameters of TT-DCNN used for ASCADv1_f dataset

| Layers | Parameters |
|---|---|
| **Conv1D_1** | Number of filters/channels out = 60, kernel size = 50, stride = 1, padding = 0. |
| **Average Pooling** | kernel size = 50, stride = 100. |
| **LTT Block Layer** | Number of filters/channels out = 120, kernel size = 7, strides = 1, amplification parameter $\tau = 4$, group = 60. |
| **Linear Regression** 1 | features out = 20 |
| **Linear Regression** 2 | features out = 20 |
| **Linear Regression** 3 | features out = 256 |

**Table 4:** Sample points for each literals of $TTSCA_{big}$ on ASCADv1_f.

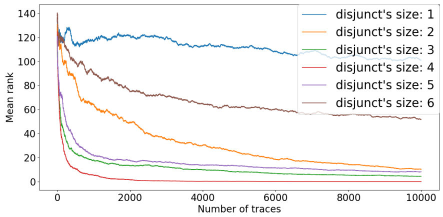
| **Sample Points** | 0 to 99 | 100 to 199 | 200 to 299 | 300 to 399 | 400 to 499 | 500 to 599 | 600 to 699 |
|---|---|---|---|---|---|---|---|
| **Literal** | $x_0$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ |

We proceed with the steps proposed in Section 3. The execution time for each steps is reported in Appendix A.4.

1. Firstly, we sieve the disjuncts based on their sizes (see Figure 11a). We observe that the only disjuncts' size where $GE = 0$ is 4 (see the red line in Figure 11a).

2. Next, we generate a list of unique CoL among all the size 4 disjuncts, denote as $Lst^{(ASCAD\_f)}_{unique}$. There are 35 of such CoLs found. We replace the original disjuncts of TT-DCNN with disjuncts of a given CoL and compute its guessing entropy. We repeat this process for each CoL found in $Lst^{(ASCAD\_f)}_{unique}$. This is illustrated in Figure 22 of Appendix A.3, and we observe that for each CoL found in $Lst^{(ASCAD\_f)}_{unique}$ their $GE \neq 0$. This suggests that our trained $TTSCA_{big}$ requires more than one CoL to retrieve the secret key. Therefore, we use Algorithm 1 to obtain a list of the critical CoLs for key recovery. Table 11b presents the critical CoLs acquired from Algorithm 1.

3. Finally, we apply the individual trimming algorithm found in Section 3 (see Figure 12a). Notice that when the literals $(x_1), (x_2), (x_4), (x_5)$ and $(x_6)$ are being trimmed independently, they all have $GE \geq 1$ (see the orange, green, purple, brown, and pink lines, respectively). When trimming $(x_1)$ and $(x_4)$ individually, their $GE$ are relatively close to 0 (i.e, $GE = 4.62$ for trimming $(x_1)$ and $GE = 1.92$ when trimming $(x_4)$). This suggest that maybe both requires more traces to attain $GE = 0$ and might not be relevant in recovering the key.



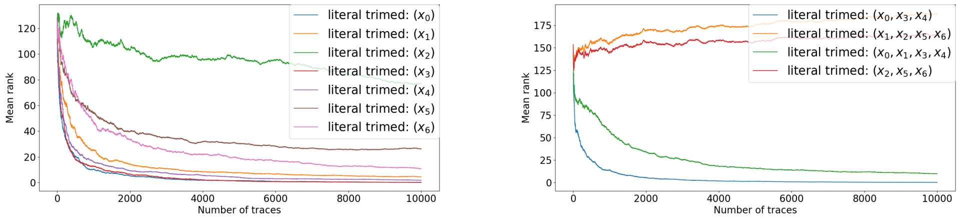
**(a)** Guessing entropy of different disjuncts' sizes.

| List Of Critical CoLs | |
|---|---|
| $(x_0, x_3, x_4, x_5)$ | $(x_0, x_1, x_2, x_3)$ |
| $(x_2, x_3, x_5, x_6)$ | $(x_0, x_1, x_2, x_5)$ |
| $(x_0, x_3, x_5, x_6)$ | $(x_1, x_2, x_4, x_6)$ |
| $(x_0, x_3, x_4, x_6)$ | $(x_0, x_4, x_5, x_6)$ |
| $(x_0, x_1, x_2, x_4)$ | $(x_3, x_4, x_5, x_6)$ |
| $(x_1, x_2, x_3, x_4)$ | $(x_1, x_2, x_4, x_5)$ |

**(b)** List of critical CoLs obtained after Algorithm 1.

**Figure 11:** Guessing entropy of step 1 and the list obtained after Algorithm 1 for $TTSCA_{big}$ on ASCADv1_f.

Thus, we first verify if $x_4$ is necessary for key recovery by trimming the literals $(x_1, x_2, x_5, x_6)$ and $(x_0, x_3, x_4)$ separately from the disjuncts with critical CoLs (see orange and blue line in Figure 12b). We see that $GE = 0$ when $(x_0, x_3, x_4)$ are trimmed (i.e., disjuncts left have literals $x_1, x_2, x_5$ and $x_6$), suggesting that the literal $x_4$ is not necessary for key recovery. Similarly, we trim $(x_2, x_5, x_6)$ and $(x_0, x_1, x_3, x_4)$ separately to check if $x_1$ is required for key recovery (see red and green line in Figure 12b). We observe that trimming $(x_2, x_5, x_6)$ results in $GE \approx 9$, implying that $x_1$ is indeed relevant in retrieving the secret key. In a nutshell, the literals required for key recovery are $x_1, x_2, x_5$ and $x_6$.



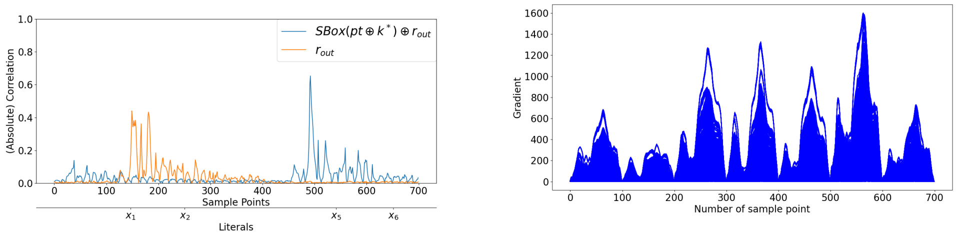
**(a)** Trimming with every individual literals (describe by the individual trimming algorithm in Section 3).

**(b)** Trimming to verify the relevancy of $x_1$ and $x_4$.

**Figure 12:** Guessing entropy for $TTSCA_{big}$ on ASCADv1_f for the step 3.

We observe that these 4 literals represent the position of PoIs in ASCADv1_f. From Table 4, the literals $x_1$ represents 100 to 199 sample points and $x_2$ represents 200 to 299 sample points which is where the PoIs of the share $r_{out}$ are (see orange line Figure 13a) while $x_5$ and $x_6$ represented 500 to 599 and 600 to 699 sample points respectively, where the PoIs of the share $SBox(pt \oplus k^*) \oplus r_{out}$ are located (see blue line in Figure 13a). Therefore, we conclude that $TTSCA_{big}$ indeed learned the position of the leakages and also a function of the shares $r_{out}$ and $SBox(pt \oplus k^*) \oplus r_{out}$ based on the literals $x_1, x_2, x_5$ and $x_6$. While Figure 13a is obtained with knowledge of the key, the neural network learns the PoIs without knowledge of the key.

To compare with previous works, we apply the explainability techniques GV on $TTSCA_{big}$ (see Figure 13b) and observe that it is difficult to tell us which points in which $TTSCA_{big}$ learns; as it differs vastly from CPA. This is because $TTSCA_{big}$ overfits the dataset since we did not use any early stopping unlike [MDP19] which managed to differentiate the PoIs of ASCADv1_f after using early stopping when training their DNN (refer to Appendix A.5 for GV of non-overfitting $TTSCA_{big}$ model). However, our method only extracts intervals of sample points that the $TTSCA_{big}$ uses. In cases where the



**(a)** CPA on the attack traces of ASCADv1_f.

**(b)** GV of $TTSCA_{big}$ on ASCADv1_f.

**Figure 13:** CPA of ASCADv1_f and GV of $TTSCA_{big}$.

evaluator wants a precise value of each sample point about the leakage that the DNN is using, GV or features map [ZBHV19] are preferred as these methods give a specific value to each sample point. Nonetheless, our work shows the exact window of PoIs that $TTSCA_{big}$ used to retrieve the secret key despite not using early stopping and even provides the DNF formulae that $TTSCA_{big}$ used for key recovery without assuming that the neural network is differentiable, unlike GV. The literals $x_1, x_2, x_5$ and $x_6$ also reveal to us the critical decision that $TTSCA_{big}$ used to retrieve the secret key on both seen and unseen traces, providing us a global interpretation.

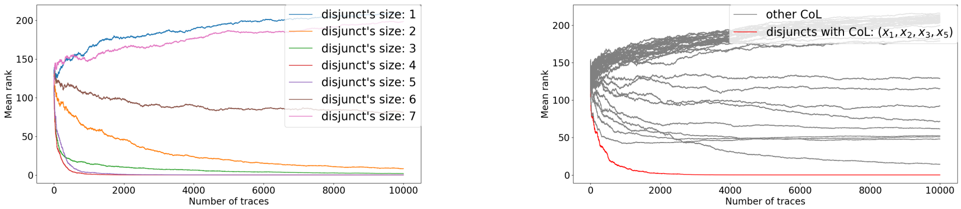
The optimal number of literals that a $TTSCA_{big}$ can learn from the sequential leakage found in ASCADv1_f is two. One literal represents the leakage from $SBox(pt \oplus k^*) \oplus r_{out}$ and the other literal represent the leakage of $r_{out}$. Therefore, the literals of $TTSCA_{big}$ that we showed in this section may not be optimal as it requires four literals to obtain the secret key instead of two literals. In the following paragraph, we report a modified miniature $TTSCA_{big}$. This modified $TTSCA_{big}$ uses only an XOR gate of two literals for key recovery. One reason why the literals we found are not optimal may be due to the heuristic nature of our methods. Therefore, finding the miniature set of disjuncts for key recovery is still an open question.

**An Interesting Result.** When manipulating the architecture from *Wouters et al.*'s work [WAGP20], we managed to train a modified $TTSCA_{big}$ (with a padding of 25 in the first Conv1D) that gives us a miniature network. Each literal represents the sample points of the traces stated in Table 5. We proceed by sieving the disjuncts based on their size. We observe that the smallest disjunct size with $GE = 0$ is 4 (red line of Figure 14a). Next, we found that there are 35 unique CoLs of size 4, and observed that the only critical CoL is the CoL for $(x_1, x_2, x_3, x_5)$ (red line of Figure 14b). There only five disjuncts (filter 66 has two disjuncts) with CoL for $(x_1, x_2, x_3, x_5)$ and is presented in the middle column of Table 6. We notice that when $(x_1)$ or $(x_5)$ are trimmed, $GE \neq 0$ (see blue and red line in Figure 15a), but when trimming $(x_2, x_3)$, we attain $GE = 0$ (see the purple line in Figure 15a). The disjuncts after trimming $(x_2, x_3)$ are presented in the last column of Table 6. By continuing to sieve the filters, we observe that filter 66 is only filter in which $GE = 0$ (see Figure 15b). The two disjuncts in filter 66 correspond to an XOR gate between $x_1$ and $\neg x_5$: $x_1 \oplus \neg x_5 = (x_1 \wedge x_5) \vee (\neg x_1 \wedge \neg x_5)$, which is the masking countermeasure protecting the underlying Sbox execution. However, it is not trivial to always learn the exact function of the countermeasure, which is the best case result we achieved. As the weights are initialised randomly, neural network trained on same dataset may differ and thus the function learned would differ.

*To summarize, the modified $TTSCA_{big}$ managed to learn a miniature set of rules with just one preprocessing block, one XOR gate (i.e., the masking function), and one linear regression to retrieve the key.*

**Table 5:** Sample points for each literals for the TT-DCNN trained with padding of 25.

| **Sample Points** | 0 to 74 | 75 to 174 | 175 to 274 | 275 to 374 | 375 to 474 | 475 to 574 | 575 to 674 |
|---|---|---|---|---|---|---|---|
| **Literal** | $x_0$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ |



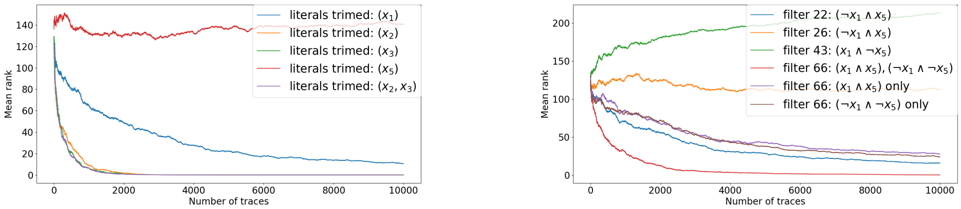
**(a)** Separated by different disjunct sizes for ASCADv1_f dataset.

**(b)** Individual CoL for disjuncts with size 4.

**Figure 14:** Guessing entropy obtained when applying step 1 and 2 on the modified $TTSCA_{big}$ trained on ASCADv1_f.

**Table 6:** Disjuncts with CoL for $(x_1, x_2, x_3, x_5)$ before and after trimming $(x_2, x_3)$ using the modified $TTSCA_{big}$.

| **Filter Number** | **Disjuncts Before Trim** | **Disjuncts After Trim** |
|---|---|---|
| 22 | $(\neg x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_5)$ | $(\neg x_1 \wedge x_5)$ |
| 26 | $(\neg x_1 \wedge x_2 \wedge x_3 \wedge x_5)$ | $(\neg x_1 \wedge x_5)$ |
| 43 | $(x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_5)$ | $(x_1 \wedge \neg x_5)$ |
| 66 | $(x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_5), (\neg x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_5)$ | $(x_1 \wedge x_5), (\neg x_1 \wedge \neg x_5)$ |



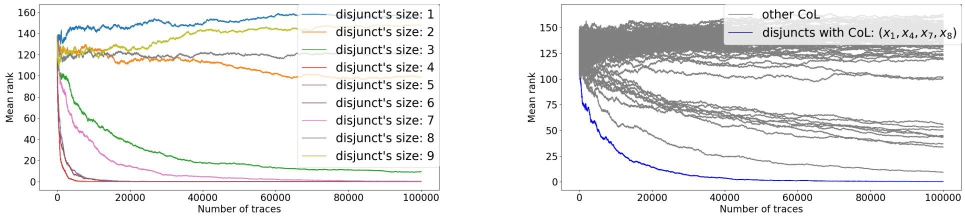
**(a)** Different literals trimmed (before trimming $(x_2, x_3)$).

**(b)** The guessing entropy of each filter (after trimming $(x_2, x_3)$).

**Figure 15:** Guessing entropy of the CoL for $(x_1, x_2, x_3, x_5)$ obtained from the modified $TTSCA_{big}$ (i.e., before and after trimming $(x_2, x_3)$).

**Interpretation of the DNF Equations of $TTSCA_{big}$ on ASCADv1_r.** There are 1400 sample points for ASCADv1_r, and we change the Conv1D_1 to have a kernel size of 75 and the average pooling to have a kernel size of 75 with stride 150. This gives us $TTSCA_{big}$ with a patch size of 9 where each literal $x_q$ represents the sample points between $q * 150$ and $(q+1) * 150 - 1$ for $q = 0, \ldots, 8$ which corresponds to 150 sample points of the traces.

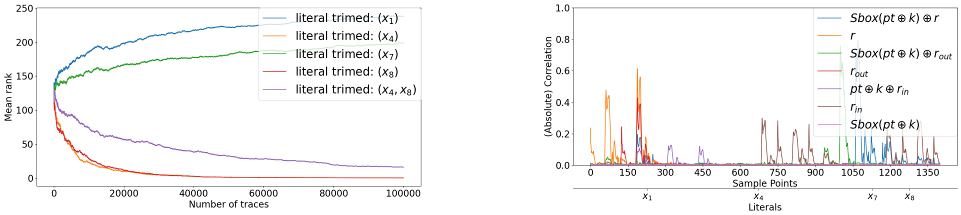
We proceed with the same methodology by sieving disjuncts based on the size first. From Figure 16a, size 4 is the smallest size with $GE = 0$. Next, there are 121 unique CoLs of size 4, and observe that one of the critical CoL is the CoL for $(x_1, x_4, x_7, x_8)$ (blue line of Figure 16b). We further apply the individual trimming algorithm on the disjuncts with CoLs of $(x_1, x_4, x_7, x_8)$. We observe from Figure 17a that $x_1$ and $x_7$ seem to be important. Therefore, we further trim to check if both $x_1$ and $x_7$ are necessary literals by trimming $(x_4, x_8)$ from disjuncts with CoLs of $(x_1, x_4, x_7, x_8)$. When we trim $(x_4, x_8)$ we



**(a)** Guessing entropy of different disjuncts' sizes.

**(b)** Individual CoL for disjuncts with size 4.

**Figure 16:** Guessing entropy of step 1 and step 2 for $TTSCA_{big}$ on ASCADv1_r.



**(a)** Trimming literals of disjuncts with CoL of $(x_1, x_4, x_7, x_8)$ for ASCADv1_r dataset.

**(b)** CPA on attack traces of ASCADv1_r.

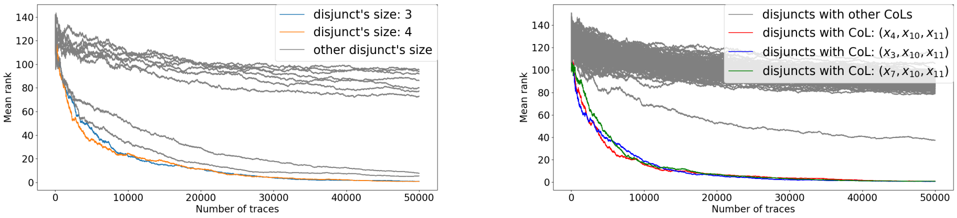
**Figure 17:** Guessing entropy for $TTSCA_{big}$ on ASCADv1_r for step 3 and CPA for ASCADv1_r.

have $GE > 0$ (see purple line in Figure 17a), but if we just trim $x_4$ or $x_8$ individually the $GE = 0$. This means that whenever $x_4$ is trimmed away, $x_8$ contains enough information to obtain $GE = 0$, and vice versa. If we compare with CPA, we see that the literal $x_1$ could be associated to several leakage, that is $r, r_{out}, Sbox(pt \oplus k) \oplus r$ and $pt \oplus k \oplus r_{in}$, while the literal $x_7, x_8$ could correspond to the leakages $r_{in}$ and $Sbox(pt \oplus k) \oplus r$ (see Figure 17b). In addition, $x_4$ could represent leakage of $r_{in}$. As a result, we show that $TTSCA_{big}$ uses three literals either $(x_1, x_7, x_8)$ or $(x_1, x_4, x_7)$ to recover the key. Thus, an evaluator can obtain positions of PoIs where the $TTSCA_{big}$ learns in a random key setting.

### 4.2.2 AES_HD_ext

Next, we validate our approach in a low SNR setting; hence we target traces from FPGA. We focus on the AES_HD_ext dataset, which is an extension of the AES_HD dataset. It consists of 500k traces each with 1250 sample points. The main leakage comes from the register writing in the last round $SBox^{-1}(ct_{11} \oplus k_{15}) \oplus ct_{11}$ where $ct_j$ is the $j^{th}$ byte of the ciphertext, $k_{15}$ is the 15th byte of the last round key and $SBox^{-1}$ is the AES inverse Sbox. The SNR observed from [ZBHV19] is 0.01554. We use 50k traces for the profiling phase and another 50k for the attack phase.

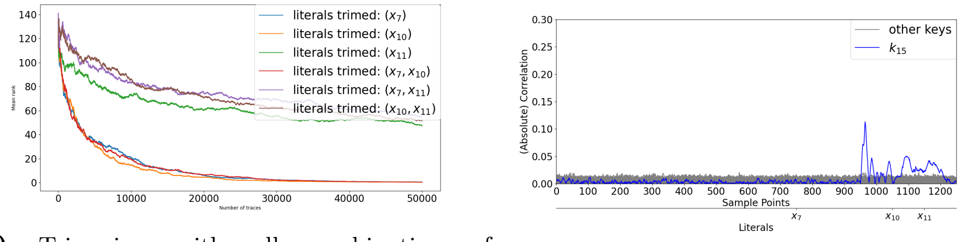
**Interpretation of the DNF Equations of $TTSCA_{big}$ on AES_HD_ext:** Since there are 1250 sample points, the $TTSCA_{big}$ used here have the same Conv1D_1 as the one use in ASCADv1_f and therefore having a patch size of 12 with each literal $x_q$ represent the sample points between $q * 100$ and $(q + 1) * 100 - 1$ for $q = 0, \dots, 11$. As before, we first sieve the disjuncts based on their size (see Figure 18a) and observe that size 3 is the smallest disjunct's size (see blue line in Figure 18a). We proceed with the



**(a)** Different disjuncts' sizes.

**(b)** Individual CoL of size 3.

**Figure 18:** Guessing entropy after seiving disjuncts based on their sizes and each CoL individual of size 3 for AES_HD_ext.



**(a)** Trimming with all combination of $(x_7, x_{10}, x_{11})$.

**(b)** CPA on attack traces of AES_HD_ext.

**Figure 19:** Guessing entropy after trimming and CPA of AES_HD_ext.

next step by first generating a list of unique CoL from the disjuncts with a size equal to 3, denoted as $Lst_{unique}^{(AES\_HD)}$. There are a total of 127 CoLs with size 3 found. Then we replace the original disjuncts of $TTSCA_{big}$ with a given CoL and compute its guessing entropy. We repeat this process for each CoL found in $Lst_{unique}^{(AES\_HD)}$ independently. The CoLs with $GE = 0$ are $(x_3, x_{10}, x_{11})$, $(x_4, x_{10}, x_{11})$ and $(x_7, x_{10}, x_{11})$ as depicted in Figure 18b. All three CoLs provided similar results, thus we shall focus only on $(x_7, x_{10}, x_{11})$. Since there are only 3 literals in the CoL for $(x_7, x_{10}, x_{11})$, we exhaust all possible scenario for trimming (excluding trimming all literals). This is illustrated in Figure 19a. Whenever $x_{11}$ is trimmed, we attain $GE \neq 0$. In fact, if $x_{11}$ is used on its own (i.e., trim literals $(x_7, x_{10})$) it result in $GE = 0$. This reveals that the literal $x_{11}$ itself is necessary for recovering the secret key. We compare our results with CPA on AES_HD_ext dataset. The literal $x_{11}$ corresponds to the PoIs found in 1100 to 1200 sample points of the traces (see Figure 19b). This implies that our technique also works in a low SNR hardware-implemented setting.

## 4.3 Limitation

The first limitation of using $TTSCA_{big}$ is that it requires more traces to retrieve the secret key compared to general CNN, but this is at the expense of interpreting what the neural network learns. We compare this trade-off between interpretability and efficiency by showing the number of attack traces needed by $TTSCA_{big}$ and the state-of-the-arts DL-SCA models for ASCADv1_f and AES_HD_ext in Table 7. In other words, $TTSCA_{big}$ is recommended if interpretation is the primary objective rather than efficiency. We also note that Algorithm 1 requires a significant amount of time to run (see Table 8 in Appendix A.4). Therefore, one should also consider the time required for analysis when using $TTSCA_{big}$ for interpretation.

Secondly, our method may not always find the most miniature set of disjuncts for key

**Table 7:** DLSCA benchmark of $N\bar{T}_{GE}$ value depending on datasets.

| | **CNN** | **Ours,** $TTSCA_{big}$ |
|---|---|---|
| **ASCADv1 (fixed key, no desync)** | 191 [ZBHV19] | 7222 |
| **AES_HD_ext** | 831 [ZXF+19] | 50000 |

recovery as it is heuristic in nature. Therefore, it is still an open question to find the optimal set of disjuncts that the $TTSCA_{big}$ learns. Thirdly, there is a hard limit to the number of literals that can be handled due to usage of Quine–McCluskey algorithm leading to low scalability to really large real traces. Current proposal is to use larger pooling windows in order overcome the hard limit. However, increasing pooling window may results in the interpretation of large interval of sample points to be not interesting (low resolution of PoI detections). Moreover, larger pooling windows might introduce more noise leading to an unsuccessful attack. In such case, using GV or feature maps in [ZBHV19] might be more suitable as these methods gives interpretation of each sample points. Furthermore, $TTSCA_{big}$ does not currently work on jitter/desynchronization countermeasure. We leave the extension of $TTSCA_{big}$ to these countermeasures for subsequent work. Lastly, other explainability techniques can be used on any DNN, but our methods are only applicable to the family of TT-DCNN. Nevertheless, our proposed methodology gives us a glimpse of what this family of TT-DCNN learns in the SCA context.

## 5 Conclusion

In this work, we apply the interpretable neural network called the Truth Table Deep Convolutional Neural Networks (TT-DCNNs) [BPKY22] in the context of SCA. The TT-DCNN can convert the neural network into SAT equations (in the form of DNF), which allows us to interpret what it has learned leading us to peek into the black-box. We proposed two different TT-DCNN-based architectures, namely $TTSCA_{small}$ and $TTSCA_{big}$, where special adjustments are made to $TTSCA_{big}$ to work with real traces. We proposed a methodology to analyze the DNF equations in the context of SCA. Our experiments show that both $TTSCA_{small}$ and $TTSCA_{big}$ indeed use the PoIs to retrieve the secret key based on their SAT formulae. These formulae retrieve the device's secret key even on traces that are not observed, giving us a global interpretation of proposed TT-DCNN-based neural networks. Application to desynchronization/jitter is left for future work.

## References

[AARR03] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM Side—Channel(s). In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 29–45, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[ANS19] André Araujo, Wade Norris, and Jack Sim. Computing Receptive Fields of Convolutional Neural Networks. *Distill*, 2019. https://distill.pub/2019/computing-receptive-fields.

[BBM+15] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PloS one*, 10(7):e0130140, 2015.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, pages 16–29, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[Bla38] Archie Blake. Corrections to Canonical expressions in Boolean algebra. *Journal of Symbolic Logic*, 3(2):112–113, 1938.

[BLR12] Timo Bartkewitz and Kerstin Lemke-Rust. Efficient template attacks based on probabilistic multi-class support vector machines. In *International Conference on Smart Card Research and Advanced Applications*, pages 263–276. Springer, 2012.

[BPKY22] Adrien Benamira, Thomas Peyrin, and Bryan Hooi Kuen-Yew. Truth-table net: A new convolutional architecture encodable by design into sat formulas, 2022.

[BPS+20] Ryad Benadjila, Emmanuel Prouff, Rémi Strullu, Eleonora Cagli, and Cécile Dumas. Deep learning for side-channel analysis and introduction to ASCAD database. *J. Cryptogr. Eng.*, 10(2):163–188, 2020.

[Bra78] R.N. Bracewell. *The Fourier Transform and its Applications.* McGraw-Hill Kogakusha, Ltd., Tokyo, second edition, 1978.

[CDP17] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. Convolutional Neural Networks with Data Augmentation Against Jitter-Based Countermeasures. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 45–68, Cham, 2017. Springer International Publishing.

[CNHR18] Chih-Hong Cheng, Georg Nührenberg, Chung-Hao Huang, and Harald Ruess. Verification of binarized neural networks via inter-neuron factoring. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 279–290. Springer, 2018.

[CRR03] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template Attacks. In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 13–28, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[DBN$^+$01] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Advanced Encryption Standard (AES), 2001-11-26 2001.

[DV16] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[EST$^+$22] Maximilian Egger, Thomas Schamberger, Lars Tebelmann, Florian Lippert, and Georg Sigl. A Second Look at the ASCAD Databases. In Josep Balasch and Colin O'Flynn, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 75–99, Cham, 2022. Springer International Publishing.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[GBR22] Anupam Golder, Ashwin Bhat, and Arijit Raychowdhury. Exploration into the Explainability of Neural Network Models for Power Side-Channel Analysis. In *Proceedings of the Great Lakes Symposium on VLSI 2022*, pages 59–64, 2022.

[GHO15] Richard Gilmore, Neil Hanley, and Maire O'Neill. Neural network based attack on a masked implementation of AES. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 106–111. IEEE, 2015.

[HCS$^+$16] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. *Advances in neural information processing systems*, 29, 2016.

[HGDM$^+$11] Gabriel Hospodar, Benedikt Gierlichs, Elke De Mulder, Ingrid Verbauwhede, and Joos Vandewalle. Machine learning in side-channel analysis: a first study. *Journal of Cryptographic Engineering*, 1(4):293–302, 2011.

[HGG19] Benjamin Hettwer, Stefan Gehrer, and Tim Güneysu. Deep neural network attribution methods for leakage analysis and symmetric key recovery. Cryptology ePrint Archive, Report 2019/143, 2019. https://eprint.iacr.org/2019/143.

[HZ12] Annelie Heuser and Michael Zohner. Intelligent machine homicide. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 249–264. Springer, 2012.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, 2015.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[JR20] Kai Jia and Martin Rinard. Efficient Exact Verification of Binarized Neural Networks. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1782–1795. Curran Associates, Inc., 2020.

[KB17] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, 2017.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, page 388–397, Berlin, Heidelberg, 1999. Springer-Verlag.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[KUMH17] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-Normalizing Neural Networks, 2017.

[LBM14] Liran Lerman, Gianluca Bontempi, and Olivier Markowitch. Power analysis attack: an approach based on machine learning. *International Journal of Applied Cryptography*, 3(2):97–115, 2014.

[LBM15] Liran Lerman, Gianluca Bontempi, and Olivier Markowitch. A machine learning approach against a masked AES. *Journal of Cryptographic Engineering*, 5(2):123–139, 2015.

[LPB$^+$15] Liran Lerman, Romain Poussier, Gianluca Bontempi, Olivier Markowitch, and François-Xavier Standaert. Template attacks vs. machine learning revisited (and the curse of dimensionality in side-channel analysis). In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 20–33. Springer, 2015.

[LSL$^+$20] Xiao-Hui Li, Yuhan Shi, Haoyang Li, Wei Bai, Yuanwei Song, Caleb Chen Cao, and Lei Chen. Quantitative evaluations on saliency methods: An experimental study, 2020.

[Lue21] Knud Lasse Lueth. State of the IoT 2020: 12 billion IoT connections, surpassing non-IoT for the first time, 2021.

[MDP19] Loïc Masure, Cécile Dumas, and Emmanuel Prouff. Gradient Visualization for General Characterization in Profiling Attacks. In Ilia Polian and Marc Stöttinger, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 145–167, Cham, 2019. Springer International Publishing.

[MGH14] Amir Moradi, Sylvain Guilley, and Annelie Heuser. Detecting hidden leakages. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *Applied Cryptography and Network Security*, pages 324–342, Cham, 2014. Springer International Publishing.

[MPP16] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking Cryptographic Implementations Using Deep Learning Techniques. pages 3–26, 12 2016.

[PBP20] Guilherme Perin, Ileana Buhan, and Stjepan Picek. Learning when to stop: a mutual information approach to fight overfitting in profiled side-channel analysis. Cryptology ePrint Archive, Report 2020/058, 2020. https://eprint.iacr.org/2020/058.

[PPM+21] Stjepan Picek, Guilherme Perin, Luca Mariot, Lichao Wu, and Lejla Batina. Sok: Deep learning-based physical side-channel analysis. Cryptology ePrint Archive, Report 2021/1092, 2021. https://eprint.iacr.org/2021/1092.

[PR13] Emmanuel Prouff and Matthieu Rivain. Masking against Side-Channel Attacks: A Formal Security Proof. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 142–159, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[PRB09] Emmanuel Prouff, Matthieu Rivain, and Régis Bevan. Statistical analysis of second order differential power analysis. *IEEE Transactions on computers*, 58(6):799–811, 2009.

[PWP22] Guilherme Perin, Lichao Wu, and Stjepan Picek. I know what your layers did: Layer-wise explainability of deep learning side-channel analysis. Cryptology ePrint Archive, Paper 2022/1087, 2022. https://eprint.iacr.org/2022/1087.

[RM21] Olivier Roussel and Vasco Manquinho. Pseudo-Boolean and cardinality constraints. In *Handbook of satisfiability*, pages 1087–1129. IOS Press, 2021.

[SMY09] François-Xavier Standaert, Tal G. Malkin, and Moti Yung. A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009*, pages 443–461, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[ST17] Ravid Shwartz-Ziv and Naftali Tishby. Opening the Black Box of Deep Neural Networks via Information. *CoRR*, abs/1703.00810, 2017.

[ST18] Leslie N. Smith and Nicholay Topin. Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates, 2018.

[STK+17] Daniel Smilkov, Nikhil Thorat, Been Kim, Fernanda Viégas, and Martin Wattenberg. SmoothGrad: removing noise by adding noise, 2017.

[STY17] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *International conference on machine learning*, pages 3319–3328. PMLR, 2017.

[SXW+20] Hongguang Sun, Conglei Xu, Yudan Wu, Shuangdai Zou, and Fang Wan. *L-CNN: An Improved Convolutional Neural Network to Capture Long-Distance Dependencies*, pages 119–128. 01 2020.

[Tim19] Benjamin Timon. Non-Profiled Deep Learning-based Side-Channel attacks with Sensitivity Analysis. volume 2019, page 107–131, Feb. 2019.

[TPB00] Naftali Tishby, Fernando C. Pereira, and William Bialek. The information bottleneck method, 2000.

[UVSV06] C. Umans, T. Villa, and A.L. Sangiovanni-Vincentelli. Complexity of two-level logic minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(7):1230–1246, 2006.

[vdVPB21] Daan van der Valk, Stjepan Picek, and Shivam Bhasin. Kilroy Was Here: The First Step Towards Explainability of Neural Networks in Profiled Side-Channel Analysis. In Guido Marco Bertoni and Francesco Regazzoni, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 175–199, Cham, 2021. Springer International Publishing.

[WAGP20] Lennert Wouters, Victor Arribas, Benedikt Gierlichs, and Bart Preneel. Revisiting a Methodology for Efficient CNN Architectures in Profiling Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):147–168, Jun. 2020.

[WWJ$^+$21] Lichao Wu, Yoo-Seung Won, Dirmanto Jap, Guilherme Perin, Shivam Bhasin, and Stjepan Picek. Explain some noise: Ablation analysis for deep learning-based physical side-channel analysis. Cryptology ePrint Archive, Report 2021/717, 2021. https://eprint.iacr.org/2021/717.

[WZLW19] Zhuo Wang, Wei Zhang, Ning Liu, and Jianyong Wang. Transparent Classification with Multilayer Logical Perceptrons and Random Binarization, 2019.

[WZLW21] Zhuo Wang, Wei Zhang, Ning Liu, and Jianyong Wang. Scalable Rule-Based Representation Learning for Interpretable Classification, 2021.

[YHPC17] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent Trends in Deep Learning Based Natural Language Processing. *CoRR*, abs/1708.02709, 2017.

[ZBC$^+$22] Gabriel Zaid, Lilian Bossuet, Mathieu Carbone, Amaury Habrard, and Alexandre Venelli. Conditional Variational AutoEncoder based on Stochastic Attack. Cryptology ePrint Archive, Report 2022/232, 2022. https://ia.cr/2022/232.

[ZBHV19] Gabriel Zaid, Lilian Bossuet, Amaury Habrard, and Alexandre Venelli. Methodology for Efficient CNN Architectures in Profiling Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(1):1–36, Nov. 2019.

[ZF13] Matthew D Zeiler and Rob Fergus. Visualizing and Understanding Convolutional Networks, 2013.

[ZXF$^+$19] Libang Zhang, Xinpeng Xing, Junfeng Fan, Zongyue Wang, and Suying Wang. Multi-label Deep Learning based Side Channel Attack. In *2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 1–6, 2019.

# A Appendix

## A.1 Inner Working of TT-DCNN for ASCADv1_f

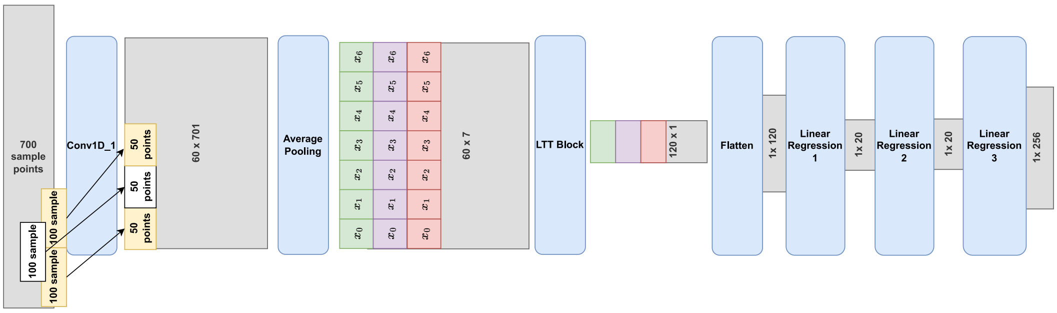
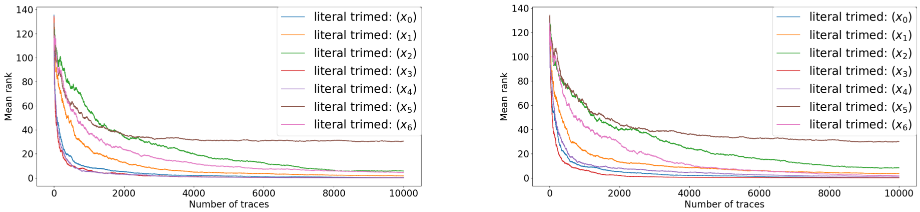


**Figure 20:** Inner working of TT-DCNN for ASCADv1_f

Figure 20 shows the inner working of our TT-DCNN architecture proposed for ASCADv1_f. We can observe that each literal represents a 100 sample point of the traces.

## A.2 Trimming Operation of $TTSCA_{big}$ Before Any Preprocessing or Before Separating the Disjuncts Based Their CoL on ASCADv1_f.



**(a)** Trimming before sieving disjuncts based on their size (before step 1).

**(b)** Trimming before separating disjuncts based on their CoLs (right before step 2).

**Figure 21:** Guessing entropy for $TTSCA_{big}$ on ASCADv1_f before step 1 and right before step 2 of proposed methodology.

Figure 21 shows the guessing entropy on ASCADv1_f when trimming individual literals before sieving the disjuncts based on their size or their CoL. In Section 4.2.1, it was concluded that the important literals are $x_1, x_2, x_5$ and $x_6$. However, when trimming literals $(x_1), (x_2)$ and $(x_6)$ individually before the first two steps, the guessing entropy remains relatively close to 0, which could just mean that it requires more traces to attack (see Figure 21a and Figure 21. Therefore, if we apply the trimming operation before either of the first two steps, we will miss out on the literals $x_1, x_2$ and $x_6$ as the critical literals for retrieving the secret key.

## A.3 Guessing entropy of Each Individual CoL of $TTSCA_{big}$ on ASCADv1_f.

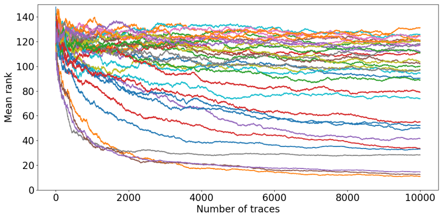


**Figure 22:** Guessing entropy of each individual CoL of $TTSCA_{big}$ from disjuncts of size equal to 4

Figure 22 shows the guessing entropy of each CoL as describe in step 2 of Section 3. We observe that none of the CoL managed to recover the key with 15k traces. The smallest guessing entropy obtained is $\approx 10$. This means that $TTSCA_{big}$ combines CoLs to recover the key. Therefore, we use Algorithm 1 to obtain the list of critical CoLs.

## A.4 Execution time of the Analysis of $TTSCA_{big}$ on ASCADv1_f.

In this section, we provide the readers the execution time for each steps when applying $TTSCA_{big}$ on ASCADv1_f (see Table 8). The execution timing is used for 7 literals with 10000 attack traces of ASCADv1 fixed key running on single NVIDIA-GeForce-GTX-970.

**Table 8:** Execution time of each steps of $TTSCA_{big}$ on ASCADv1_f.

| Steps | Time (mins) |
|---|---|
| Training | 6.59 |
| SAT-Conversion | 0.09 |
| Sieveing disjuncts based on their size | 30.91 |
| Separate disjuncts based on their CoLs | 156.94 |
| Trimming disjuncts based on the literals | 30.76 |

## A.5 Gradient Visualization of non-overfitting $TTSCA_{big}$ model on ASCADv1_f.

We trained $TTSCA_{big}$ over 200 epochs and compute its GE every 5 epoch and pick the smallest epoch with $GE = 0$. The $TTSCA_{big}$ trained with 20 epoch is the smallest epoch that obtains a $GE = 0$. We run GV on our non-overfitting $TTSCA_{big}$ and depict in Figure 23.

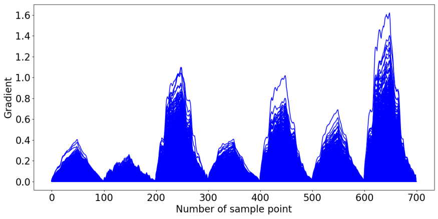


**Figure 23:** GV of non-overfitting $TTSCA_{big}$ on ASCADv1_f.

From Figure 23, we observe that the PoIs are not very visible. However, we consider the two highest peaks as the PoIs even if there are not the only peaks. This provides us with valuable information about the leakages, which is in contrast to the overfitting case, where nothing can be concluded from GV (see Figure 13b). We shall give some reasons why the PoIs are not as easy to observe. Firstly, GV is unstable [LSL$^+$20]. This means that the explanations change drastically with small perturbations to the input. Secondly, the choice of the hyperparameter for the average pool, which resulted in the features' importance are by windows of sample points as in Figure 23, may contribute to the decreased visibility of the PoIs.